Planos de execução curiosos

Abr 16 Publicado por **saziba** em **GPO Blogs**

Analisar planos de execução deve ser uma atividade recorrente na vida de um desenvolvedor Oracle.

Para sair um pouco do padrão, separei alguns planos com algumas operações diferentes dos típicos "TABLE ACCESS FULL", "INDEX UNIQUE SCAN".

Bora lá?

```
set autot trace explain
```

No primeiro caso utilizei uma tabela chamada "objetos" com uma coluna "object_id" como chave primária.
Imagine o típico cenário onde uma pesquisa é feita e um parâmetro passado pode ser nulo e, neste caso, o parâmetro deve ser ignorado.
Nesse cenário temos um impasse, se o parâmetro for passado, a cardinalidade da query diminuiria e poderemos utilizar um determinado índice, no entanto, se o parâmetro não for passado a cardinalidade da query aumentaria significativamente e um "FULL TABLE SCAN" poderia ser a melhor opção.

Para simular isso devemos criar uma variável:

```
variable b number;
```

Uma das alternativas é construir a seguinte query:

```
select * from objetos where object_id = :b or :b is null;
```

... e obtermos o plano:

```
Execution Plan
----------------------------------------------------------
Plan hash value: 4071356626

-------------------------------------------------------------------------------
| Id  | Operation         | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |         |   720 | 64800 |    56   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| OBJETOS |   720 | 64800 |    56   (0)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   1 - filter(:B IS NULL OR "OBJECT_ID"=TO_NUMBER(:B))
```

Outra alternativa seria:

```
select * from objetos where object_id = nvl(:b,object_id);
```

... e então o plano ficaria:

```
Execution Plan
----------------------------------------------------------
Plan hash value: 1003103769

-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |         | 14378 |  1263K|    58   (0)| 00:00:01 |
|   1 |  CONCATENATION                |         |       |       |            |          |
|*  2 |   FILTER                      |         |       |       |            |          |
|*  3 |    TABLE ACCESS FULL          | OBJETOS | 14377 |  1263K|    56   (0)| 00:00:01 |
|*  4 |   FILTER                      |         |       |       |            |          |
|   5 |    TABLE ACCESS BY INDEX ROWID| OBJETOS |     1 |    90 |     2   (0)| 00:00:01 |
|*  6 |     INDEX UNIQUE SCAN         | OBJ_PK  |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(:B IS NULL)
   3 - filter("OBJECT_ID" IS NOT NULL)
   4 - filter(:B IS NOT NULL)
   6 - access("OBJECT_ID"=:B)
```

Exatamente o que precisávamos.

Para os que não a conhecem eu vos apresento a operação "CONCATENATION".
É como se o Oracle reescrevesse a nossa query da seguinte forma:

```
select * from objetos where object_id = :b and :b is not null
UNION ALL
select * from objetos where object_id is not null and :b is null;
```

Dessa forma ele consegue um "duplo" plano de execução aplicando os filtros e os acessos adequados para cada situação.

No nosso exemplo, o predicado é aplicado na PK, mas é bom saber que essa situação não é conveniente quando se tem uma coluna nullable e queremos preservar as correspondências nulas quando o parâmetro não é passado.

Agora vou mostrar alguns casos peculiares utilizando outras estruturas que não são exatamente tabelas no banco de dados.

Veja a operação que acontece quando se usa uma nested table instanciada em tempo de execução:

```
select * from table(sys.odcinumberlist(1,2,3));

Execution Plan
----------------------------------------------------------
Plan hash value: 1748000095

--------------------------------------------------------------------------------------------
| Id  | Operation                             | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                      |      |  8168 | 16336 |    29   (0)| 00:00:01 |
|   1 |  COLLECTION ITERATOR CONSTRUCTOR FETCH|      |  8168 | 16336 |    29   (0)| 00:00:01 |
--------------------------------------------------------------------------------------------
```

É um pouco diferente da nested table sendo consultada depois de instanciada, como acontece em processos PLSQL.
Veja o que acontece quando pedimos o plano de execução da própria DBMS_XPLAN.DISPLAY

```
select * from table(dbms_xplan.display);

Execution Plan
----------------------------------------------------------
Plan hash value: 2137789089

---------------------------------------------------------------------------------------------
| Id  | Operation                           | Name    | Rows  | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |         |  8168 | 16336 |    29   (0)| 00:00:01 |
|   1 |  COLLECTION ITERATOR PICKLER FETCH| DISPLAY |  8168 | 16336 |    29   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------
```

Apesar de esquisitas essas operações são bem simples de entender: O SQL Engine está acessando estruturas de memória da própria PGA.

Um pouco de engenharia reversa, agora:

Quem já viu esse plano de execução?

```
Execution Plan
----------------------------------------------------------
Plan hash value: 4223128547

--------------------------------------------------------------------------------
| Id  | Operation            | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |      |  8168 | 16336 |    29   (0)| 00:00:01 |
|   1 |  XMLTABLE EVALUATION |      |       |       |            |          |
--------------------------------------------------------------------------------
```

Essa é a query que o originou:

```
select * from xmltable('xml' passing xmltype('<xml>1</xml>') columns num number path '/');
```

Quem ainda não brincou com XMLTABLEs estou preparando um post específico sobre esse assunto.

Para o próximo caso, vou preparar o modelo criando uma tabela com duas colunas com valores únicos, id1 e id2, e índices únicos criados nelas

```
create table obj as select rownum id1, rownum id2, lpad('#',4000,'#') v from dual connect by level <= 10000;

Table created.


create unique index obj_idx1  on obj (id1);

Index created.


create unique index obj_idx2  on obj (id2);

Index created.


exec dbms_stats.gather_table_stats(USER,'OBJ');
```

```
PL/SQL procedure successfully completed.
```

Depois de coletar as estatísticas vou tentar selecionar somente as colunas indexadas e realizar um "INDEX FAST FULL SCAN"

```
select id1, id2 from obj;
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 730912574

--------------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      | 10000 | 80000 |  2753   (1)| 00:00:34 |
|   1 |  TABLE ACCESS FULL| OBJ  | 10000 | 80000 |  2753   (1)| 00:00:34 |
--------------------------------------------------------------------------------
```

Hmmm... não deu certo pois, o FFS só acontece quando desconsideramos valores nulos. Então deixe-me dizer claramente ao Oracle que essas colunas não aceitam nulos:

```
alter table obj modify(id1 number not null);

Table altered.

alter table obj modify(id2 number not null);

Table altered.
```

Tentando de novo:

```
select id1, id2 from obj;
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 63238214

-------------------------------------------------------------------------------------------
| Id  | Operation             | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |                  | 10000 | 80000 |    42   (0)| 00:00:01 |
|   1 |  VIEW                 | index$_join$_001 | 10000 | 80000 |    42   (0)| 00:00:01 |
|*  2 |   HASH JOIN           |                  |       |       |            |          |
```

```
|   3 |    INDEX FAST FULL SCAN| OBJ_IDX1         | 10000 | 80000 |    26   (0)| 00:00:01 |
|   4 |    INDEX FAST FULL SCAN| OBJ_IDX2         | 10000 | 80000 |    26   (0)| 00:00:01 |
---------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(ROWID=ROWID)
```

Aí está. Dois "INDEX FAST FULL SCAN" na mesma tabela e então realizou o JOIN dos dois índices. Veja o nome do objeto criado internamente para isso "index$_join$_001".

Vou reconstruir a tabela e agora não vou colocar a constraint NOT NULL

```
drop table obj;

Table dropped.

create table obj as select rownum id1, rownum id2, lpad('#',4000,'#') v from dual connect by level <= 10000;

Table created.


create unique index obj_idx1  on obj (id1);

Index created.


create unique index obj_idx2  on obj (id2);

Index created.


exec dbms_stats.gather_table_stats(USER,'OBJ');

PL/SQL procedure successfully completed.
```

Agora, se a query disser que não deseja verificar valores nulos, obtemos o mesmo INDEX_JOIN:

```
select id1, id2 from obj where id1 is not null and id2 is not null;
```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 63238214

-------------------------------------------------------------------------------------------------
| Id  | Operation              | Name             | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT       |                  | 10000 | 80000 |    42   (0)| 00:00:01 |
|   1 |  VIEW                  | index$_join$_001 | 10000 | 80000 |    42   (0)| 00:00:01 |
|*  2 |   HASH JOIN            |                  |       |       |            |          |
|*  3 |    INDEX FAST FULL SCAN| OBJ_IDX1         | 10000 | 80000 |    26   (0)| 00:00:01 |
|*  4 |    INDEX FAST FULL SCAN| OBJ_IDX2         | 10000 | 80000 |    26   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access(ROWID=ROWID)
   3 - filter("ID1" IS NOT NULL)
   4 - filter("ID2" IS NOT NULL)
```

```
drop table obj;

Table dropped.
```

Para encerrar esse post, o plano de execução mais curioso que já vi.
Vejamos o que acontece quando usamos a cláusula ROLLUP no GROUP BY:

```
select * from dual group by rollup (dummy);

Execution Plan
----------------------------------------------------------
Plan hash value: 2656630603

-----------------------------------------------------------------------------------
| Id  | Operation                  | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |      |     1 |     2 |     3  (34)| 00:00:01 |
```

```
|   1 |  SORT GROUP BY NOSORT ROLLUP|      |     1 |     2 |     3  (34)| 00:00:01 |
|   2 |   TABLE ACCESS FULL         | DUAL |     1 |     2 |     2   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
```

Nada de mais, certo?
Agora veja o que acontece quando colocamos duas vezes a mesma coluna no ROLLUP:

```
select * from dual group by rollup (dummy,dummy);

Execution Plan
----------------------------------------------------------
Plan hash value: 189552792


-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                     | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                          |     1 |     2 |     9  (12)| 00:00:01 |
|   1 |  TEMP TABLE TRANSFORMATION    |                          |       |       |            |          |
|   2 |   MULTI-TABLE INSERT          |                          |       |       |            |          |
|   3 |    SORT GROUP BY NOSORT ROLLUP|                          |     1 |     2 |     3  (34)| 00:00:01 |
|   4 |     TABLE ACCESS FULL         | DUAL                     |     1 |     2 |     2   (0)| 00:00:01 |
|   5 |    DIRECT LOAD INTO           | SYS_TEMP_0FD9D66B3_366A0 |       |       |            |          |
|   6 |    DIRECT LOAD INTO           | SYS_TEMP_0FD9D66B4_366A0 |       |       |            |          |
|   7 |   VIEW                        |                          |     3 |     6 |     6   (0)| 00:00:01 |
|   8 |    VIEW                       |                          |     3 |     6 |     6   (0)| 00:00:01 |
|   9 |     UNION-ALL                 |                          |       |       |            |          |
|  10 |      TABLE ACCESS FULL        | SYS_TEMP_0FD9D66B3_366A0 |     1 |     2 |     2   (0)| 00:00:01 |
|  11 |      TABLE ACCESS FULL        | SYS_TEMP_0FD9D66B3_366A0 |     1 |     2 |     2   (0)| 00:00:01 |
|  12 |      TABLE ACCESS FULL        | SYS_TEMP_0FD9D66B4_366A0 |     1 |     2 |     2   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------------------------
```

Alguém pode me explicar isso?

Toda essa parafernalha de coisas tem um motivo: evitar ter que sumarizar os registros duas vezes.
O o Oracle cria duas tabelas temporárias, uma pro group by (que ele chamou de "SYS_TEMP_0FD9D66B**3**_366A0") e outra pro rollup ("SYS_TEMP_0FD9D66B**4**_366A0 ") e utiliza um MULTI TABLE INSERT para carregá-las, para depois expor o resultset a partir delas mostrando os dados do group by, quantas vezes for necessário.
Louco né?

Eu gostaria de falar do BITMAP CONVERSION TO/FROM ROWID, mas para deixar o blog mais interativo vou pedir que alguém me mande um exemplo com a criação de um modelo de dados para o estudo de caso. Pode ser?

Por hoje é só
Espero que tenham gostado desse post.

```
exit
```

TAGs: **diversão**, **Explain**, **Explain Plan**, **Plan table**, **plano de execuçao**, **sql**, **sql avançado**, **sql tuning**

- **PRINT**